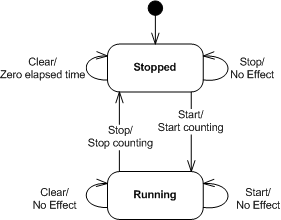
# Exercises

For the first three questions, consider a stopwatch as described here. The state machine below presents the desired behavior.



It has the following hardware.

* Buttons for start, stop and clear functions.
  + Pressing Start starts the stopwatch running. If pressed multiple times, stopwatch continues running without resetting elapsed time.
  + Pressing Stop stops the stopwatch from counting.
  + Pressing Clear zeroes out the elapsed time if the stopwatch is not running. If it is running, the clear button is ignored.
* A timer which triggers an interrupt every 1 ms. The timer drives a counter which counts milliseconds since system start-up, and can be read as elapsed\_time\_counter.
* A display to show elapsed time with 1 ms resolution. The display must be updated 10 times per second.

1. Design pseudocode for the software using event-triggered scheduling with interrupts. Assume that each button can generate an interrupt.

* Use a variable called state to indicate whether the stopwatch is stopped or running
* Use a variable called elapsed\_time to track how much time has elapsed since the start button was pressed.
* Use a variable called display\_delay to track how many milliseconds remain until the display needs to be updated again.

Main thread:

state = stopped

display\_delay = 100

elapsed\_time = 0

Start ISR:

state = running

Timer ISR:

if state == running

elapsed\_time += 1 ms

display\_delay -= 1

if display\_delay == 0 {

display\_delay = 100

display elapsed\_time

}

Stop ISR:

state = stopped

Clear ISR

if state == stopped

elapsed\_time = 0

1. Now design pseudocode for the software using a **static scheduler without using any interrupts**. Assume that the timer updates a hardware register called elapsed\_time\_register every millisecond.

* Use a variable called state to indicate whether the stopwatch is stopped or running
* Use a variable called start\_time to record when the start button was pressed.
* Use a variable called stop\_time to record when the stop button was pressed.
* Use a variable called next\_display\_update to indicate when the display needs to be updated next.

state = stopped

display elapsed\_time\_counter

next\_display\_update = elapsed\_time\_counter + 100

while (1) {

if start switch pressed {

if state == stopped {

start\_time = elapsed\_time\_counter

state = running

}

}

if stop switch pressed {

if state == running {

stop\_time = elapsed\_time\_counter

state = stopped

}

}

if clear switch pressed {

if state == stopped {

start\_time = stop\_time

}

}

if elapsed\_time\_counter > next\_display\_update {

if (state == running)

display elapsed\_time\_counter - start\_time

else

display stop\_time – start\_time

next\_display\_update = next\_display\_update + 100

}

}

1. Create a flowchart to represent your solution to the previous question.



1. Consider a system with the following tasks. We wish to **minimize** the response time for task C. For each type of scheduler, describe the sequence of processing activities which will lead to the minimum and the maximum response times for task C. Assume that each task is ready to run and there are no further task releases.

|  |  |
| --- | --- |
| Task | Duration |
| A | 3 |
| B | 1 |
| C | 2 |

* 1. Static, non-preemptive scheduler

Best Case: Task C starts immediately (at time 0). Tr = 0 + 2 = 2

Worst Case: Task A and Task B run first. Tr = 0 + 3 + 1 + 2 = 6

* 1. Dynamic, non-preemptive scheduler

Best Case: Task C starts immediately (at time 0).

Worst Case: Longest task (A) just started running ε time units ago, so C won’t run until it finishes. Tr = 0 + 3 – ε + 2 = 5 - ε

* 1. Dynamic, preemptive scheduler

Best Case: Task C starts immediately (at time 0).

Worst Case: Longest task (A) just started running ε time units ago, but it is preempted by C.   
Tr = 0 + 2 = 2

1. Design a state chart showing the operation of a calculator with digits 0 through 9, four functions (+ - \* /) , a clear button, and an equals button. Assume that there are two valid sequences of operations:
2. <operand> <operator> <operand> <equals> : this performs the operation specified on the two operands in order entered, and places the result in the accumulator.
3. <operand> <equals>: this performs the operation specified on the accumulator and the operand in order entered, and places the result in the accumulator.

There are various possible state charts, based on one’s assumptions about calculator functionality. Here is one state chart:

